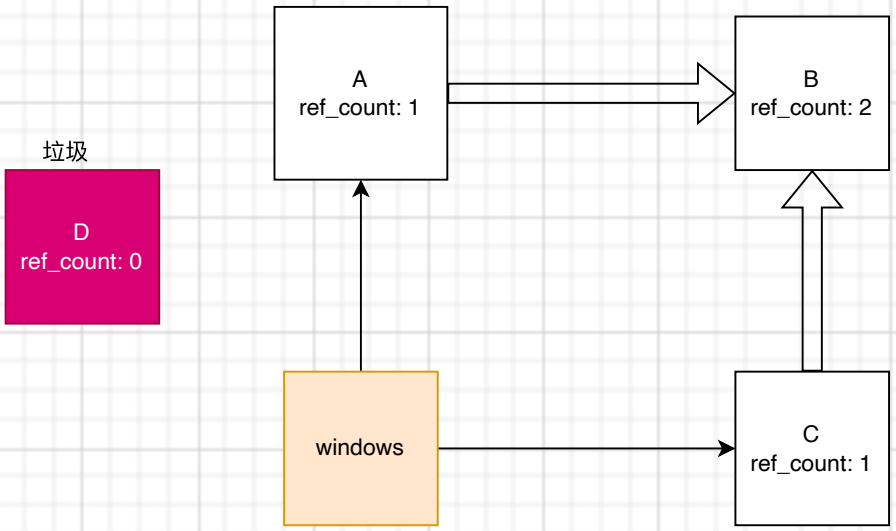
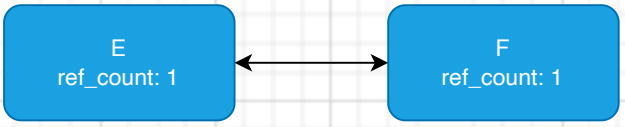
## 引用计数算法

引用计数法：每个对象上会有一个引用计数count，记录被引用的次数，当引用次数为0，会被垃圾回收器回收。




**缺点：**

无法解决循环引用的问题，即当两个对象互相引用的时候，垃圾回收器无法处理



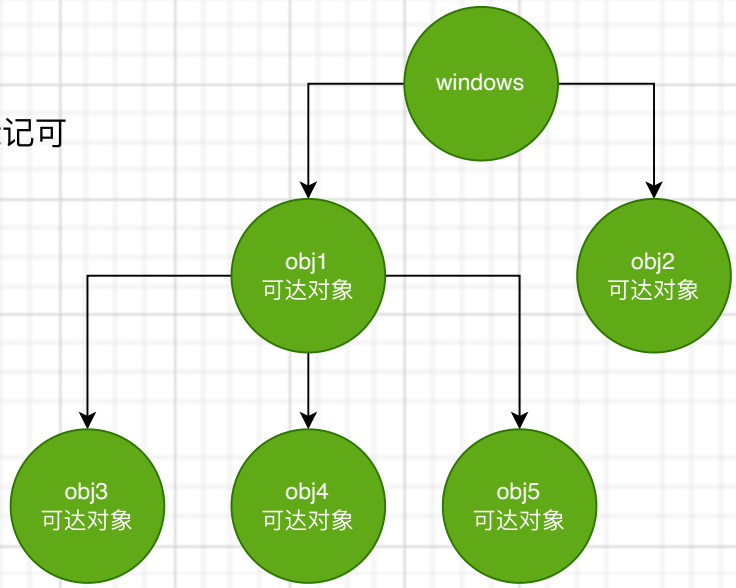

---

## 标记清除算法

阶段一：

遍历所有对象，标记可达对象。

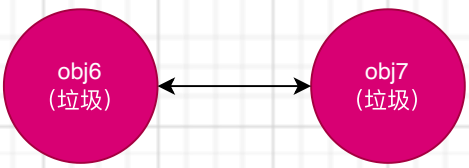
阶段二：




**优缺点**

1. 可以解决循环引用的问题

2. 会造成内存的碎片化



# 缺点

内存的碎片化，造成部分内存无法使用，如图中，白色的部分是可用空间，但是他们在内存地址空间中，并非连续的，所用造成的问题是它们没有办法被有效使用

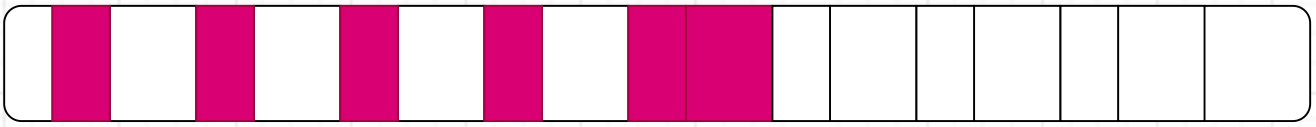
---

# 标记清除+整理

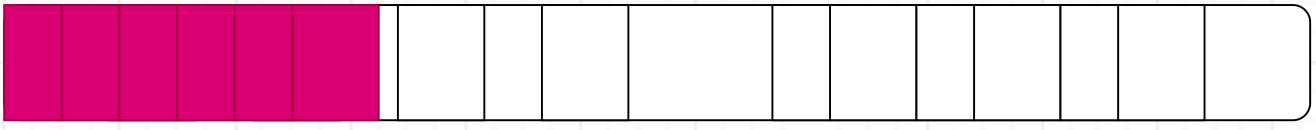
这个方法是在标记清除的基础之上添加了整理的步骤。

不过，在清除阶段，会先执行整理，移动对象的位置然后，再执行清除。使得内存的空间得以连续。

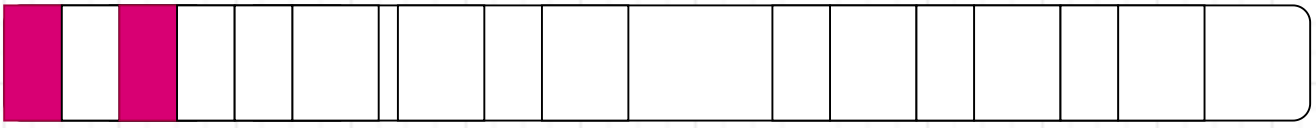
回收前



移动活动对象



释放活动空间



可以观察到，此时并不会出现大片的碎片空间

# V8 垃圾回收机制 - 空间划分

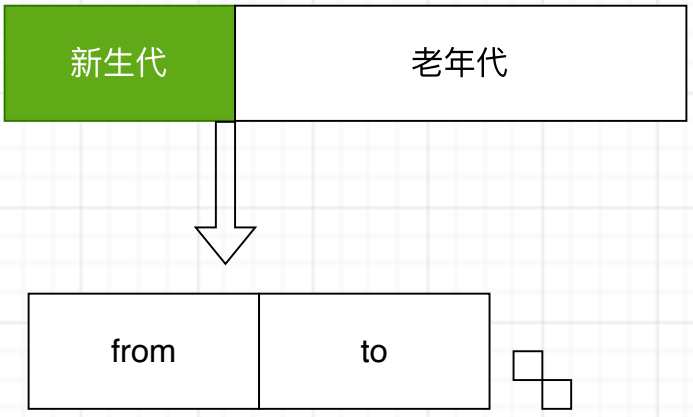
垃圾回收一般指的是回收堆内的空间内存。

V8 采用分代回收的机制：分为新生代 老年代

针对新生代和老年代，采用不同的垃圾回收算法

V8常用的算法：

1. 分代回收
2. 空间复制
3. 标记清除
4. 标记整理
5. 标记增量 （为了提升效率）


## 新生代算法

新生代内分部分分为两部分，from区 和 to区。from是正在使用的空间，to是空闲的空间。

采用的是 复制算法 + 标记整理

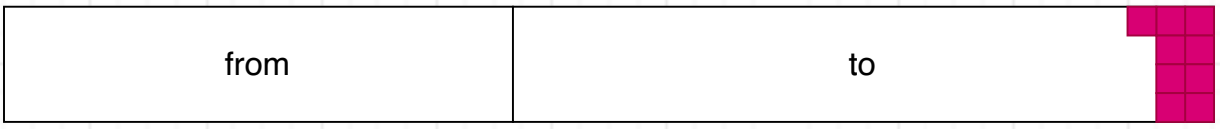
# 执行过程

实质上还是采用的是 标记清除算法，此时为整理这个过程单独设置了一个空间叫做 to。

在标记清除from区后，会将活动对象从from区挪动至to区，然后from和to互换

标记整理回收后的from区


将活动对象复制至 to区



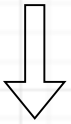

交换from与to区


---

# 新生代晋升老年代

1. 当新生代内的对象多次未被回收成功后，为了腾出新生代的空间，这些对象会晋升为老年代，***V8并未明确究竟是多少次，根据java的经验是 16次***

2. To 空间使用率超过 80%的时候，也会将它们直接送入老年代

# 老年代

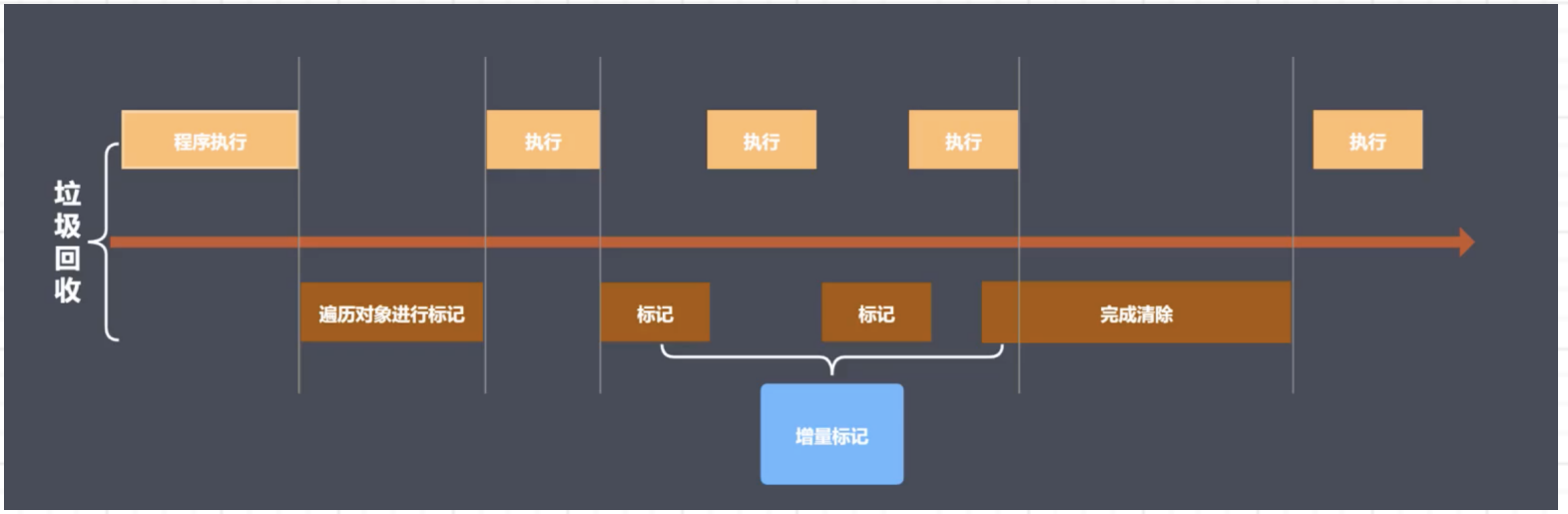
1. 老生代在 64位系统为 1.4G，32位系统位700M大小
2. 主要采用标记清除、标记整理、增量标记算法

回收前


标记清除 回收后


整理： 当有新生代晋升到老年，发现老年代空间不足的时候，会进行老年代的整理


标记增量：将一整段回收操作分为小段的操作，使程序的响应性更强

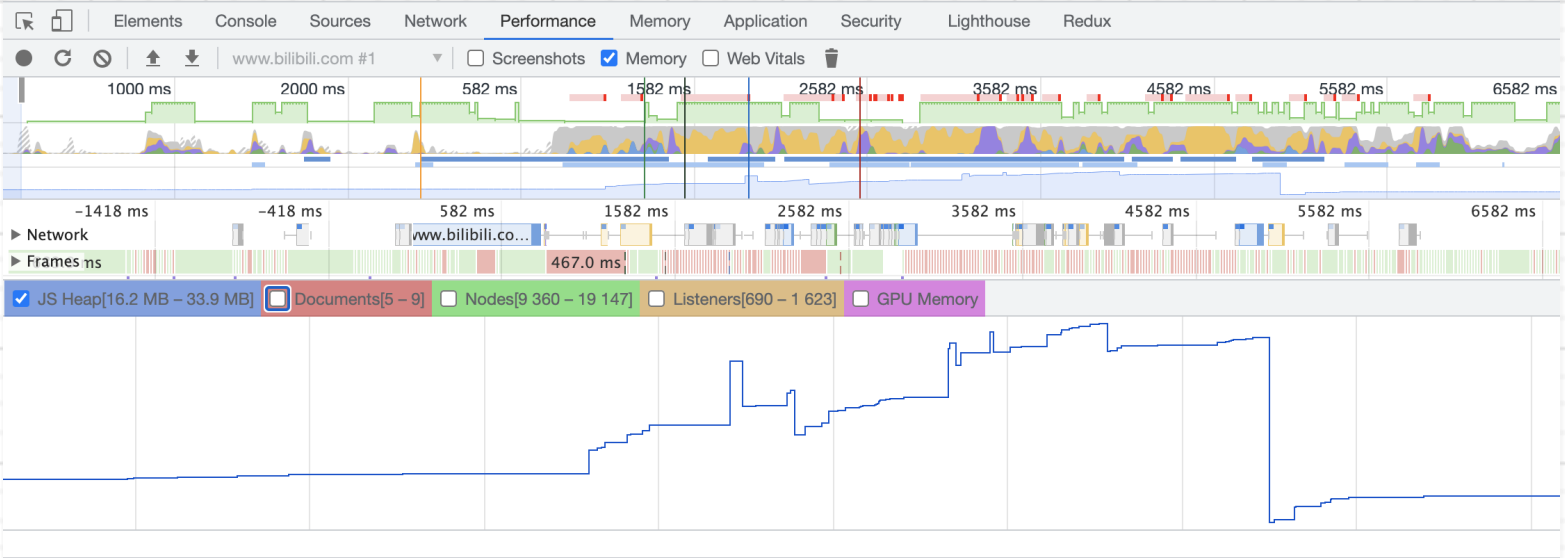


# Performance 监控内存

基本使用：

1. 打开浏览器
2. 进去 dev tool 选择性能
3. 开启录制
4. 停止录制
5. 分析内存信息

以B站为例图




---

# 问题的体现

1. 页面出现延迟加载或者经常性的暂停
    - 可能由于是频繁的垃圾回收
    - 内存瞬间被占满爆掉

2. 页面持续性出现糟糕的性能
    - 内存膨胀：页面申请的内存远超硬件设备所能提供的

3. 页面随着时间延长越来越差
    - 内存泄漏，随着时间的累积，泄漏的内存逐渐增多

# 界定内存问题的标准

内存泄漏：内存使用持续升高

内存膨胀：在多数设备上都存在性能问题

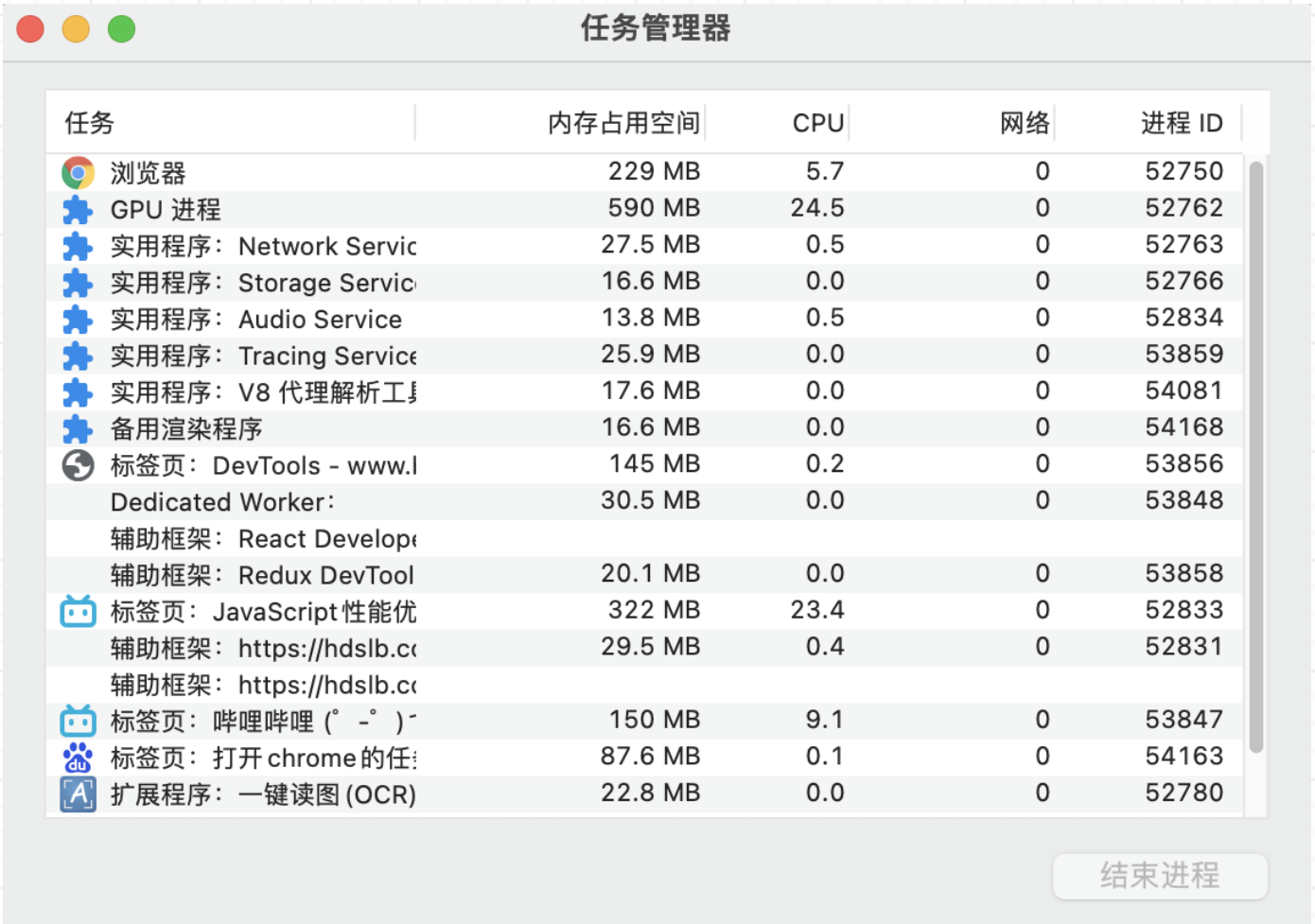
频繁的垃圾回收：通过内存变化图分析

# 监控内存的几种方式

1. 浏览器任务管理器
2. Timeline 时序图
3. 堆快照查找分离的DOM
4. 判断是否存在频繁的垃圾回收

# 浏览器任务管理器

可以观察 内存的占用情况，这个内存可以表示DOM的使用情况，如果这个内存在持续不断的膨胀，那么说明我们的程序在不停的创建新的dom



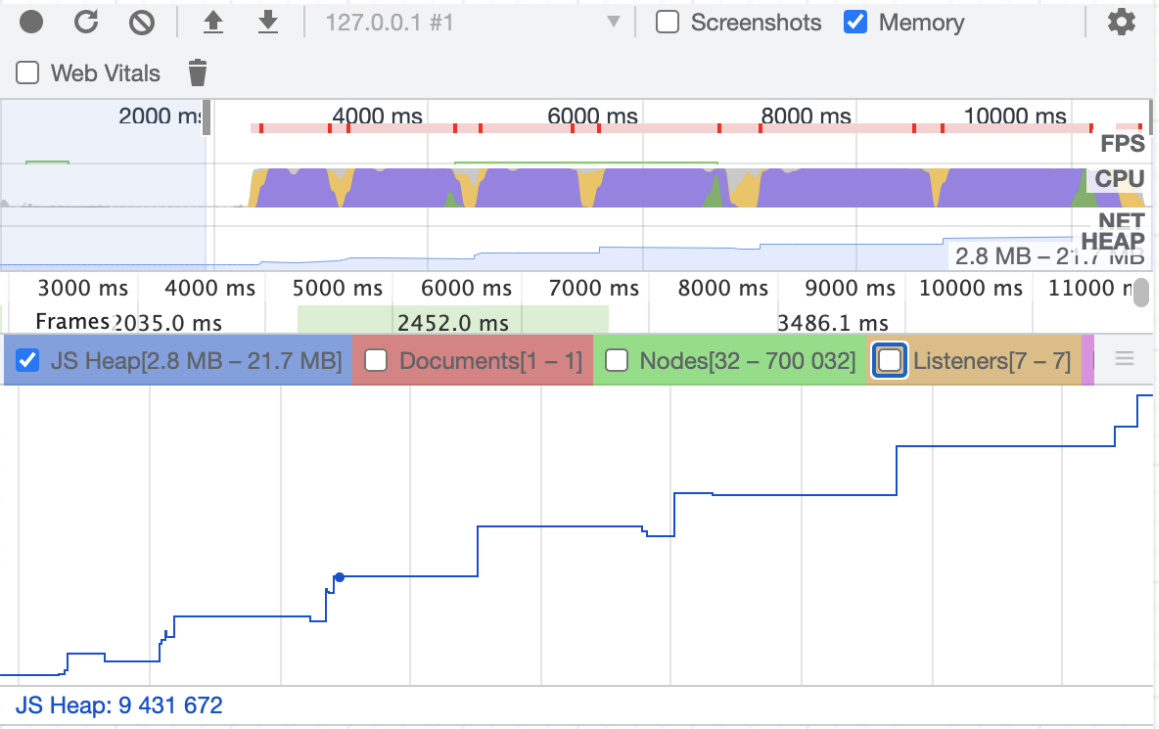
# Timeline记录内存

Timeline可以记录内存使用的变化情况。

我们可以通过“test.html”的按钮案例来做样板分析。（详细看附件）。

每当我们点击 add 按钮的时候，程序会向arr中添加大量数据。

打开timeline记录内存的变化时序图，我们可以分析出，是add 按钮会引发的内存大量占用，造成页面卡顿，进而进行优化，




# 堆快照分析 分离DOM

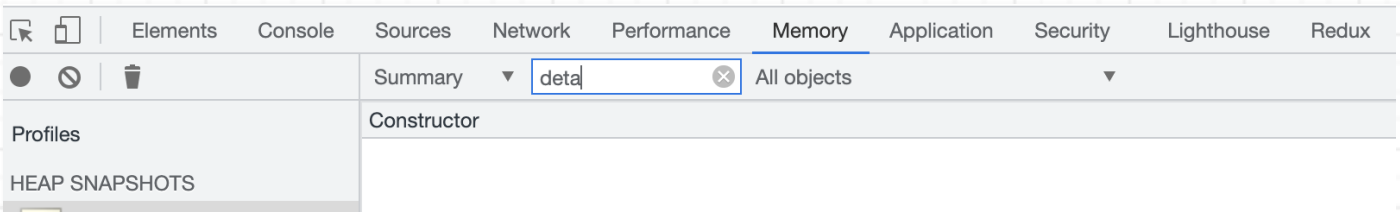
界面我们看到的元素存活在DOM树上。

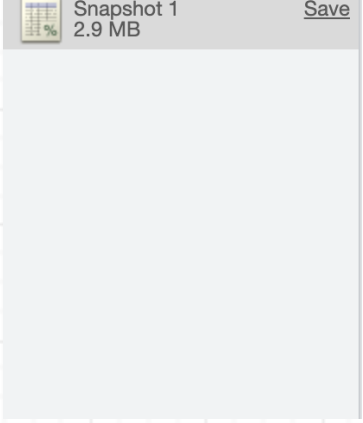
DOM节点存在不同形态：

- 垃圾DOM：当DOM对象从DOM树移除，也没有被引用

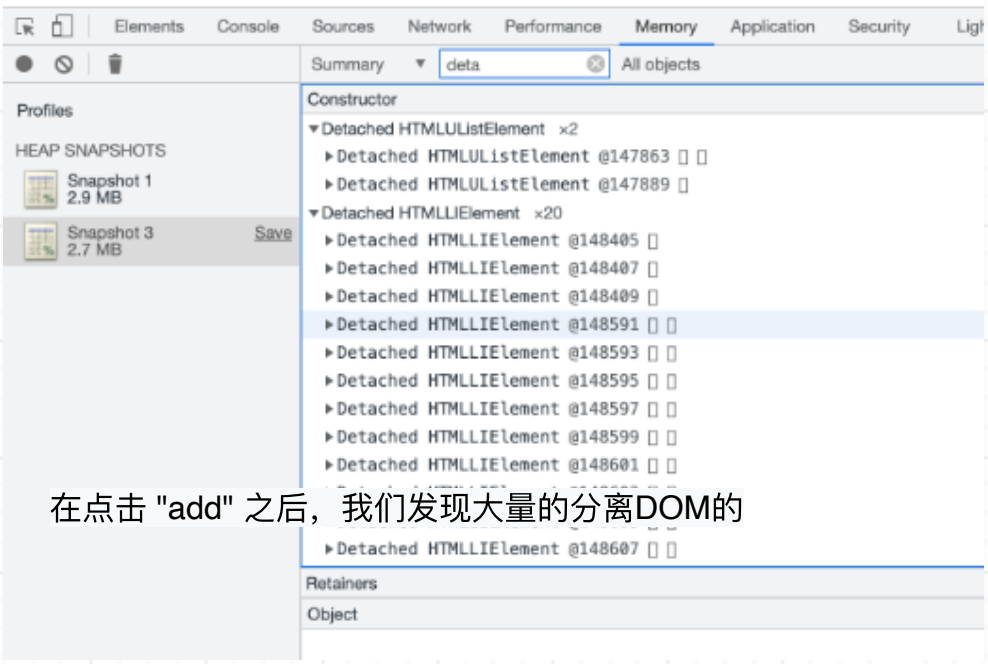
- 分离DOM：DOM节点从DOM树中移除，但是有对象引用这个DOM（此时可能造成内存泄漏）

案例 “堆快照分析.html”






在点击 "add" 之前，我们发现此时内存是没有 分离DOM的




在点击 "add" 之后，我们发现大量的分离DOM的

# 判断GC是否频繁的发生

GC 工作时 应用程序时停止的

频繁且过长的GC会导致应用假死

用户使用中感知卡顿

判断方式：

- 可以观察timeline中的内存走势，是否存在频繁快速上升了快速下降
- 任务管理器中数据频繁的增加或减少

# 主要组成部分

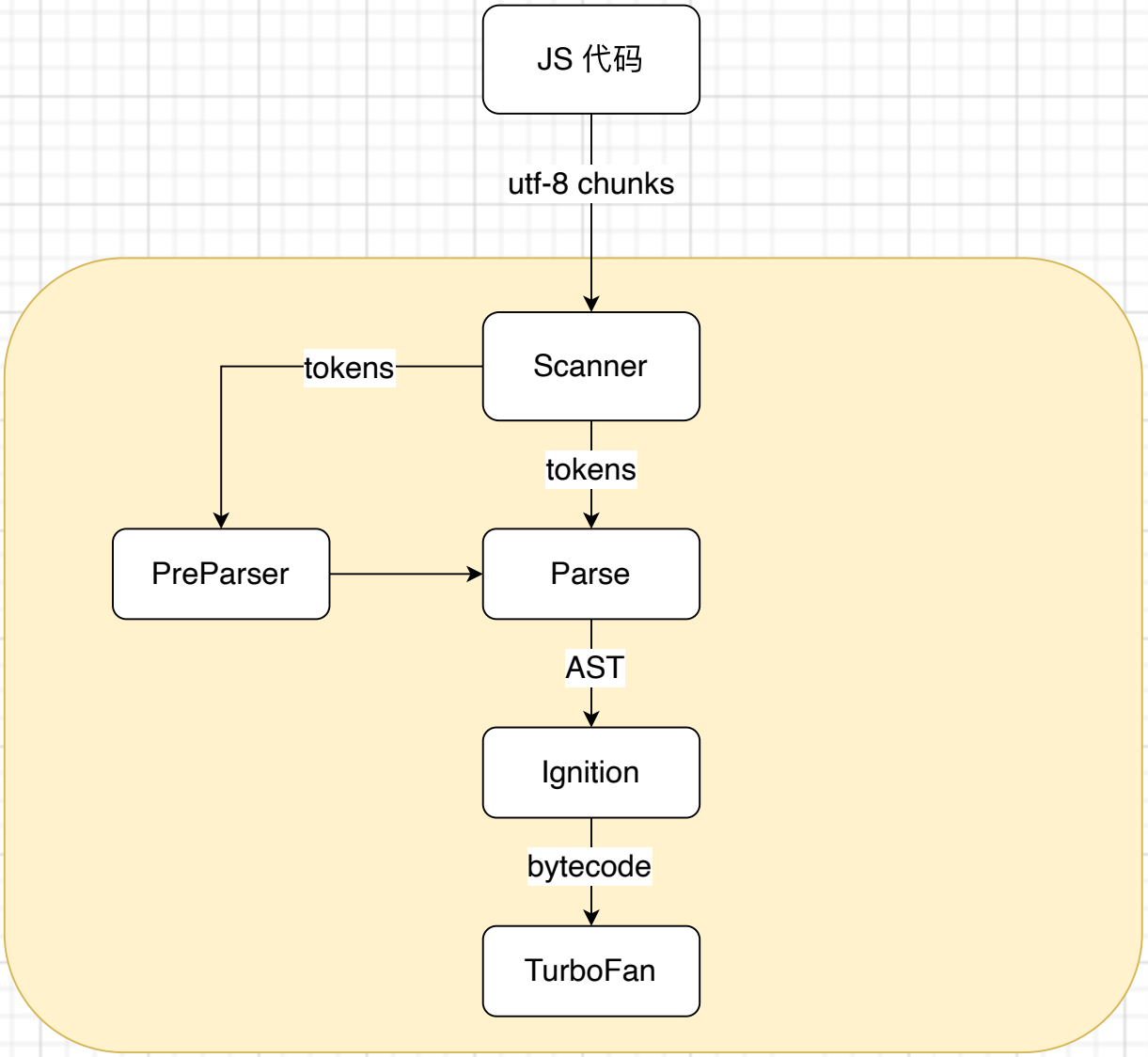
Scanner 扫描器：解析JS成token

Parser 解析器：把tokens转化成抽象语法树，也会语法的校验。

PreParse 预解析: 筛选出有效的代码片段。（比如，跳过不使用的function的声明）

Ignition 解释器：将抽象语法树转发成

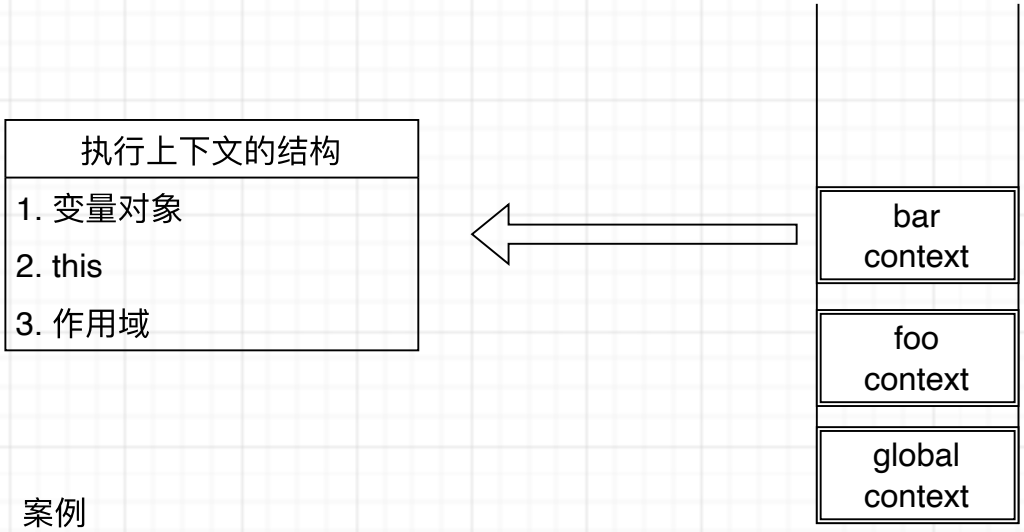
TurboFan：字节码转化为汇编代码




# 堆栈操作

# 堆栈操作

- JS 执行环境
- 执行环境盏：执行上下文的容器
- 执行上下文：每当一个函数被调用的时候，都会生成一个context推入执行上下文盏中




案例

```
bar() {}

function foo() {
        bar();
}

foo();
```